Python

Jamie Gabbay
(adapted from slides by Hans-Wolfgang Loidl)

Semester 1 2015/16

Contents

Resources

- www.python.org: official website
- Learning Python, by Mark Lutz.
- learnpythonthehardway.org/book/.
- www.python-course.eu/.
- My current favourite: docs.python.org/3.4/library/index.html. A treasure trove.

The Internet is bursting at the seams with great info on Python. Explore.

Python

- Python is named after *Monty Python's Flying Circus*
- Open source
- Highly portable
- First version was made available 1990
- Current stable version is 3.4

Python 3 vs Python 2

Python 3 offers new concepts over Python 2 (and drops some old ones).
Warning: Python 2 code samples online might not run on Python 3. A tool `python3-2to3` says what to change, and mostly works.
Common issues are:

- In Python 3, `print` is treated as any other function, especially you need to use parentheses as in write `print(x)` NOT `print x`.
- Focus on *iterators*: pattern-like functions (e.g. `map`) now return iterators, i.e. a handle to a computation, rather than a result. More on this later (e.g. `range`).

For details check:
`https://www.python.org/download/releases/3.0/`

Runtime behaviour

- Python source code is compiled to bytecode—a portable machine code optimised for rapid execution by a software interpreter.
- Compilation is performed transparently: your programs 'just run' but are actually being compiled to bytecode, and then run.
- Automatic memory management using reference counting based garbage collection. This is like Java, but unlike C where memory must be explicitly allocated and deallocated.
- No uncontrolled crash *(as in segfaults)*

You are expected to know, in general terms, about bytecode and automatic memory management. E.g. *"Q. Explain why C programs may suffer from memory leaks and Python programs do not."* or *"Python and Java compile to bytecode. Explain in general terms what the previous sentence means."* We're not looking for expertise; just general knowledge. Read up on this if you need to.

Language features

- Everything is an object (pure object-oriented design).
- Features classes and multiple inheritance.
- Higher-order functions (like ML).
- Dynamic typing and polymorphism.
- Exceptions as in Java.
- Static scoping and modules.
- Operator overloading.
- Block structure with semantic-bearing indentation (“off-side rule” as in Haskell).

Data types

- `int`, `float`, and `complex` numbers.
- Strings.
- List, tuple, and range.
- Sets and frozensets.
- Dictionaries.
- Add-on modules can define new data-types.
- Can model arbitrary data-structures using classes.

Data types

- You are expected to be familiar with the int, float, string, list, tuple, range, set, frozenset, and dictionary types.
- You are expected to be able to explain what they are and how they differ.

E.g.

- Q: Explain the difference between `set` and `frozenset`, with specific reference to the `x.issubset(y)` and `x.add(i)` methods (3 marks).
- A: `set` is mutable, `frozenset` is immutable (1 mark). Thus if `x` has `set` type then it supports both methods (1 mark), whereas if `x` has type `frozenset` then it supports `issubset` but not `add` (1 mark).

See Slide 71 and surrounding slides.

Why Python?

- Code 2-10$\times$ shorter than C#, C++, Java.
- Code is easy to comprehend.
- Good for *web scripting*.
- Scientific applications (numerical computation, natural language processing, data visualisation, etc).
- Already familiar to many of you, e.g. from the Raspberry Pi.
- Rich libraries for XML, Databases, Graphics, etc.
- *Web content management* (Zope/Plone).
- GNU Mailman
- JPython

Why Python?

- Active community
- Good libraries
- Used in famous teaching institutes (MIT, Berkeley, Heriot-Watt, etc)
- Excellent online teaching material, e.g. Online Python Tutor
- Includes advanced language features, such as functional programming.

Launching Python

- Interactive Python shell: `rlwrap python3`
- Exit with *eof* (Unix: Ctrl-D, Windows: Ctrl-Z)
- Or: `import sys; sys.exit()`
- Execute a script: `python myfile.py`

```
python3 ..python-args.. script.py ..script-args..
```

- Evaluate a Python expression

```
python3 -c "print (5*6*7)"
python3 -c "import sys; print (sys.maxint)"
python3 -c "import sys; print (sys.argv)" 1 2 3 4
```

- Executable Python script

```
#!/usr/bin/env python3
# -*- coding: iso-8859-15 -*-
```

Our first Python interaction

```
> rlwrap python3
Python 3.4.0 (default, Jun 19 2015, 14:20:21)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more informat
>>> # A comment
... x = "Hello␣World" # Let's stop faffing: set x
>>> x                 # What's x?
'Hello␣World'         # x is a string 'Hello World'
>>> print(x)          # Go on, print it
Hello World           # OK
>>> type(x)           # What's the type of x?
<class 'str'>         # A string, I told you
>>> x*3               # Copy it three times, please
'Hello␣WorldHello␣WorldHello␣World' # OK
```

Handy tip: `type(v)` yields the type of v.

Numbers

```
>>> 2+2                 # start simple
4
>>> 7/3                 # weak types;
2.3333333333333335      # return float
>>> 7/-3
-2.3333333333333335
>>> 7//3                # integer division
2
>>> 7//-3
-3
>>> 10**8               # 'To the power of'
100000000
```

More fun with numbers

```
>>> x = 7/3
>>> x.is_integer()
False
>>> x.__round__
<built-in method __round__ of float object at 0x7f9b27a35af8>
>>> x.__round__()
2
>>> type(x)
<class 'float'>
>>> type(x.__round__())
<class 'int'>
>>> x.as_integer_ratio()
(5254199565265579, 2251799813685248)
```

Floats aren't infinite precision, then!

Handy tip: Type '`x.`' then double-tab (thanks to Chris). Lists methods of `x`.

More fun with numbers

Integers are infinite precision, floats aren't. This can lead to innocent fun:

```
>>> 10**30        # integer computation
1000000000000000000000000000000
>>> int(1E30)     # float computation, convert to integer
1000000000000000019884624838656
>>> int(1E30)-10**30  # seems like rounding error crept in
19884624838656   # only 20 trillion out!
```

More fun with numbers

More innocent fun:

```
>>> def tower(n):   # Recursive function
...     if n==0:     # What does it calculate?
...         return 2
...     else:
...         return 2**(tower(n-1))
>>> tower(3)
65536
>>> tower(4)
# Answer several pages long (try it)
>>> tower(5)
# Crashes my machine
```

Python eats memory, trying to calculate `tower(5)` to infinite precision. Let's try it with floats instead . . .

More fun with numbers

```
>>> def tower(n):
...     if n==0:
...         return 2.0
...     else:
...         return 2.0**(tower(n-1))
...
>>> tower(3)
65536.0
>>> tower(4)
OverflowError: (34, 'Numerical result out of range')
```

In other words, `float` just throws up its hands and gives up.
You are required to know that integers are infinite precision in Python and floats aren't, by default, and e.g. predict the behaviour of the two `tower` functions above.

(Non-examinable aside:) Arbitrary precision float

For arbitrary precision floats, `mpmath` seems the package to use. You may need to install it
(on my machine: `apt-cache install python3-mpmath`).

```
>>> from mpmath import *
>>> mp.dps = 50; mp.pretty = True
>>> fdiv(7,3)
2.3333333333333333333333333333333333333333333333
>>> +pi
3.1415926535897932384626433832795028841971693993751
>>> +sqrt(2)
1.4142135623730950488016887242096980785696718753769
```

That's more like it!
Here's a page with 100 one-liners for calculating pi in Python:
`fredrikj.net/blog/2011/03/100-mpmath-one-liners-for-pi/`

Assignment

- Variables don't have to be declared (scripting language).

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

- Parallel assignments:

```
>>> width, height = height, width + height
```

- Short-hand notation for parallel assignments:

```
>>> x = y = z = 0  # Zero x, y and z
>>> x
0
>>> z
0
>>> x = 0 = y = z  # Worth a try
Error
```

Back to floating point

- Arithmetic operations overloaded between int and float.
- Integers converted to float on demand:

```
>>> 3 * 3.75 / .5
22.5
>>> 7. / 2
3.5
>>> float(7) / 2
3.5
```

- Exponent notation: `1e0 1.0e+1  1e-1 .1e-2`
- Typically 53 bit precision (as `double` in C).

```
>>> 1e-323
9.881312916824930e-324
>>> 1e-324
0.0
```

Back to floating point

```
>>> 1e-323
9.8813129168249309e-324
>>> 1e-324
0.0
```

If we loaded `mpmath` we can use its `power` function:

```
>>> power(10,-324)
mpf('1.0e-324')
>>> power(10,-325)
mpf('9.9999999999999997e-326')
```

Further arithmetic operations

- Remainder:

```
>>> 4 % 3
1
>>> -4 % 3
2
>>> 4 % -3
-2
>>> -4 % -3
-1
>>> 3.9 % 1.3
1.2999999999999998
```

- Division and Floor:

```
>>> 7.0 // 4.4
1.0
```

Complex Numbers

- Imaginary numbers have the suffix `j`.

```
>>> 1j * complex(0,1)
(-1+0j)
>>> complex(-1,0) ** 0.5
(6.123031769111886e-17+1j)
```

- Real- and imaginary components:

```
>>> a=1.5+0.5j
>>> a.real + a.imag
2.0
```

- Absolute value is also defined on `complex`.

```
>>> abs(3 + 4j)
5.0
```

You're not expected to know complex numbers off by heart, though in an exam I may give relevant definitions and set a question using them, just because.

Bit operations

- *Left-* (<<) and *right-shift* (>>)

```
>>> 1 << 16
65536
```

- Bitwise *and* (&), *or* (|), *xor* (^) and *negation* (~).

```
>>> 254 & 127
126
>>> 254 | 127
255
>>> 254 ^ 127
129
>>> ~0
-1
```

Bit operations

Binary representation using `bin`, go back using `int`.

```
>>> bin(1<<16)
'0b10000000000000000'
>>> type(bin(1<<16))
<class 'str'>
>>> int(bin(1<<16),2)
65536
>>> bin(0b11111110 & 0b01111111)
'0b1111110'  #lost leading zero; see below
>>> bin(0b11111110 | 0b01111111)
'0b11111111'
>>> bin(0b11111110 ^ 0b01111111)
'0b10000001'
```

What to keep leading zeroes? Use `format` instead (bit abstruse):

```
>>> format(0b11111110 & 0b01111111, '#010b')
'0b01111110'
```

Strings

- Type: `str`.

```
>> type("Hello world")
<class 'str'>
```

- Single- and double-quotes can be used

```
Input                      Output
-------                    ---------
'Python tutorial'          'Python tutorial'
'doesn\'t'                 "doesn't"
"doesn't"                  "doesn't"
'"Yes," he said.'          '"Yes," he said.'
"\"Yes,\" he said."        '"Yes," he said.'
'"Isn\'t," she said.'      '"Isn\'t," she said.'
```

Escape-Sequences

```
\\          backslash
\'          single quote
\"          double quote
\t          tab
\n          newline
\r          carriage return
\b          backspace
```

Multi-line string constants

- The expression

```
print ("This is a rather long string containing\n\
several lines of text as you would do in C.\n\
    Whitespace at the beginning of the line is\
 significant.")
```

- displays this text

```
This is a rather long string containing
several lines of text as you would do in C.
    Whitespace at the beginning of the line is significa
```

Triple-quote

- Multi-line string including line-breaks:

```
print ("""
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
""")
```

- gives

```
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to
```

Raw strings

- An `r` as prefix preserves all escape-sequences.

```
>>> print ("Hello! \n\"How are you?\"")
Hello!
"How are you?"
>>> print (r"Hello! \n\"How are you?\"")
Hello! \n\"How are you?\"
```

- Raw strings also have type `str`.

```
>>> type ("\n")
<type 'str'>
>>> type (r"\n")
<type 'str'>
```

Unicode

- Unicode-strings (own type) start with u.

```
>>> print (u"a\u0020b")
a b
>>> u"ö"
u'\xf6'
>>> type (_)
<type 'unicode'>
```

- Standard strings are converted to unicode-strings on demand:

```
>>> "this " + u"\u00f6" + " umlaut"
u'this \xf6 umlaut'
>>> print _
this ö umlaut
```

String operations

```
"hello"+"world"    "helloworld"       # concat.
"hello"*3          "hellohellohello" # repetition
"hello"[0]         "h"                # indexing
"hello"[-1]        "o"                # (from end)
"hello"[1:4]       "ell"      # slicing
len("hello")       5                  # size
"hello" < "jello"  True               # comparison
"e" in "hello"     True               # search
```

String operations

You are required to understand that string comparison is lexicographic on the underlying ASCII representation—and that this can have counter-intuitive consequences:

```
>>> "z"<"a"
False
>>> "Z"<"a"
True
>>> "Z" < "aardvark"
True
>>> "z" < "aardvark"
False
>>> "a"<"A"
False
>>> "9"<"10"
False
```

See this page for more on ‘human’ sorting.

Lists

- Lists are *mutable arrays.*

```
a = [99, "bottles of beer", ["on", "the", "wall"]]
```

- String operations work on lists.

```
a+b, a*3, a[0], a[-1], a[1:], len(a)
```

- Elements and segments can be modified.

```
a[0] = 98
a[1:2] = ["bottles", "of", "beer"]
             # -> [98, "bottles", "of", "beer",
                           ["on", "the", "wall"]]
del a[-1] # -> [98, "bottles", "of", "beer"]
```

More list operations

```
>>> a = range(5)            # [0,1,2,3,4]
>>> a.append(5)             # [0,1,2,3,4,5]
>>> a.pop()                 # [0,1,2,3,4]
5
>>> a.insert(0, 42)         # [42,0,1,2,3,4]
>>> a.pop(0)                # [0,1,2,3,4]
42
>>> a.reverse()             # [4,3,2,1,0]
>>> a.sort()                # [0,1,2,3,4]
```

You are required to understand that these methods modify the data *in-place* . . .

More list operations

. . . and that if you want the computation itself then use functions such as `reversed` or `sorted`:

```
>>> a = ['0','1' ]
>>> a.reverse()                    # a reversed in-place
>>> a
['1', '0']
>>> type(['0','1'].reverse())      # Here, ['0','1'] is reversed ...
<class 'NoneType'>                 # and garbage-collected!
>>> reversed(['0','1'])            # You probably meant this
<list_reverseiterator object at 0x7f9ea58b72b0>
>>> list(reversed(['0','1']))      # Oops.  Force execution with list
['1', '0']                         # Perfect
>>> sorted(reversed(['0','1'])) # Now sort it, just for fun
['0', '1']
```

More list operations

Really thinking about this is enough to make my head hurt. Watch this:

```
>>> a = [0,1]          # Make a list
>>> a.reverse()        # Reverse it
>>> a
[1, 0]                 # Yup, reversed
>>> type(a)            # What's its type?
<class 'list'>         # A list.
>>> a.reverse()        # Reverse it again.
>>> a = reversed(a)    # Now reverse it again ...
>>> type(a)            # What's the result type?
<class 'list_reverseiterator'>
>>> a                  # Not a list; a computation!
<list_reverseiterator object at 0x7f9ea5957f28>
>>> list(a)            # Force evaluation
[1, 0]                 # Phew!
```

Lists and strings

If `a` is a list of strings and `s` is a string, then `s.join(a)` concatenates the elements of `a` using `s` as a separator.
E.g. if `s` is a list of strings then `',`.joins`)` is a comma-separated list.
You are required to understand the code below, including the reason for the error message:

```
>>> a = [99, "bottles␣of␣beer", ["on", "the", "wall"]]
>>> a[1]+a[2]
TypeError: Can't␣convert␣'list'␣object␣to␣str␣implicitly
>>>␣a[1:2]+a[2]
['bottles of beer',␣'on',␣'the',␣'wall']
>>>␣' '.join(a[1:2]+a[2])
'bottles of beer on the wall'
>>>␣#␣Note␣list␣comprehension␣[␣exp␣for␣range␣];␣see␣later
>>>␣','.join(reversed([str(i)␣for␣i␣in␣range(11)]))+',liftoff!'
'10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, liftoff!'
```

Booleans

- `0`, `''`, `[]`, `None`, etc. are interpreted as `False`.
- All other values are interpreted as `True` (also functions!).
- `is` checks for object identity: `[] == []` is true, but `[] is []` isn't. `5 is 5` is true. More on this in Slide 75.
- Comparisons can be chained like this: `a < b == c > d`.
- The boolean operators `not`, `and`, `or` are *short-cutting* (cf ML `andalso`, `orelse`).

```
def noisy(x): print (x); return x

a = noisy(True) or noisy(False)
```

- Also works on non-Boolean values:

```
>>> '' or 'you' or 'me'
'you'
```

While

- Print all Fibonacci numbers up to 100:

```
>>> a, b = 0, 1
>>> while b <= 100:
...         print (b)
...         a, b = b, a+b
...
```

- Comparison operators: `==` `<` `>` `<=` `>=` `!=`
- **NB:** Indentation carries semantics in Python:
  - Indentation starts a block
  - De-indentation ends a block
- Or:

```
>>> a, b = 0, 1
>>> while b <= 100: print (b); a,b = b, a+b
...
```

If

Example

```
x = int(input("Please enter an integer: "))
if x < 0:
     x = -1
     print('Sign is Minus')
elif x == 0:
     print('Sign is Zero')
elif x > 0:
     print('Sign is Plus')
else:
     print('Should never see that')
```

- **NB:** `elif` instead of `else if` to avoid further indentations.

For

`for` iterates over a sequence (e.g. list, string)

Example

```
a = ['cat', 'window', 'defenestrate']
for x in a:
    print(x, len(x))
```

This prints:

```
cat 3
window 6
defenestrate 12
```

For

The iterated sequence must not be modified in the body of the loop!

Example

```
a = ['Hi','De']
for x in a: print(x,end='␣'); a.extend(a[0:1])
# This loops forever.  Run it and see.  Why?
a = ['Hi','De']
for x in a: print(x,end='␣'); a.extend(a[0:])
# This may crash your machine.  Why?
(Run it after saving all documents.)
```

In the second example, the length of a is doubled at *each cycle*.

For

We can force creation of a copy first, e.g. using segment `a[:]` or `list`.

Example

```
>>> a = ['Hi','De']
# This is dangerous
#   for x in a: print(x,end=' '); a.extend(a[0:])
# This is mostly harmless
>>> for x in a[:]: print(x,end='␣'); a.extend(a[0:])
...                          # Fingers crossed ...
Hi De                        # Phew!
>>> a                        # Why 8 elements?
['Hi', 'De', 'Hi', 'De', 'Hi', 'De', 'Hi', 'De']
>>> a = ['Hi','De']          # Reset a
>>> for x in list(a): print(x,end='␣'); a.extend(a[0:])
Hi De Hi De Hi De Hi De
>>> a
['Hi', 'De', 'Hi', 'De', 'Hi', 'De', 'Hi', 'De']
```

Range function

`range()` generates a *computation* for a range of numbers:

```
>>> range(10)
range(0, 10)          # Range from 0 unless otherwise stated
>>> type(range(10)) # What's its type?
<class 'range'>       # It's a range computation
>>> list(range(10)) # Please execute this computation
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] # OK: numbers from 0 to 9
>>> range(10)[2:4]
range(2, 3)           # Slicing doesn't trigger evaluation!
>>> list(range(10)[2:4])
[2, 3]                # List triggers evaluation.
>>> list(range(10))[2:4] # Move one tiny bracket left ...
[2, 3]                # ... now evaluates first,
                      # *then* slices (inefficient)  ...
```

Range function

```
>>> range(10**8)                # ... so take some big range
range(0, 100000000)
>>> list(range(10**8))[2:4] # Bracket one way
[2, 3]                          # Runs slow
>>> list(range(10**8)[2:4]) # Bracket the other
[2, 3]                          # Runs fast
```

`list(range(10**10))` froze my system up.
You are expected to understand that `range` does not compute a range, it creates a *computation* for a range, which may or may not be triggered.
You are expected to recognise the implications, e.g. for efficiency.

Range function

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)           # Can specify start point
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)        # Can specify jump
[0, 3, 6, 9]
>>> range(-10, -100, -30) # Also negative numbers
[-10, -40, -70]
>>> list(range(-10))       # Be careful ...
[]
>>> list(range(0,-10,-1)) # ... you probably meant this!
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Typical application: iteration over the indices of an array.

```
a = ['Mary', 'had', 'a', 'little', 'lamb']
for i in range(len(a)):
    print(i, a[i])
```

For-/While-loops: `break`, `continue`, `else`

- `break` (as in C), terminates the enclosing loop immediately.
- `continue` (as in C), jumps to the next iteration of the enclosing loop.
- The `else`-part of a loop will only be executed, if the loop hasn't been terminated using `break` construct.

Example

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print (n, 'equals', x, '*', n//x)
            break
    else: # loop completed, no factor
        print (n, 'is a prime number')
```

The empty expression

- The expression `pass` does nothing.

```
while True:
    pass # Busy ... wait for keyboard interrupt
```

- Use if an expression is syntactically required but doesn't have to do any work.

The None value

None is handy for implementing *partial functions*. Suppose you want to implement division division; what to do with divide-by-zero?
You could raise an exception . . . or just return a None value. Try these programs:

```
def mydiv(x,y):
    if y==0: return None;
    else:    return (x/y)
print(mydiv(1,0))
# output: "None"

# A day in the life of Superman:
superman_todo={1:"Get out bed",2:"Save World",3:"Breakfast"}
print(superman_todo.get(4)) # 4th item of Superman's todo?
# output: "None"
```

Procedures

- Procedures are defined using the key word `def`.

```
def fib(n):    # write Fibonacci series up to n
    """Print a Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print (b)
        a, b = b, a+b
```

- Variables `n`, `a`, `b` are local.
- The return value is `None` (hence, it is a procedure rather than a function).

```
print (fib(10))
```

More on `fib` in Slide 83.

A procedure as an object

- Procedures are values in-themselves.

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

On parameters

- Assignment to parameters of a function are local.

```
def bla(l):
    l = []

l = ['not', 'empty']
bla(l)
print(l) # Output: ['not', 'empty'], not []
```

- `l` is a reference to an object.
- The referenced object can be modified:

```
def exclamate(l):
    l.append('!')

exclamate(l)
print(l) # '!' added!
```

Global Variables

- The access to a global variable has to be explicitly declared.

```
def clear_l():
    global l
    l = []

l = ['not', 'empty']
clear_l()
print(l)
```

- ...prints the *empty* list.

Return values

- The `return` construct immediately terminates the procedure.
- The `return ...value...` construct also returns a concrete result value.

```
def fib2(n):
    """Return the Fibonacci series up to n."""
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)    # see below
        a, b = b, a+b
    return result

f100 = fib2(100)    # call it
f100                # write the result
```

Doc-strings

- The first expression in a function can be a string (as in elisp).

```
def my_function():
    """Do nothing, but document it.

    No, really, it doesn't do anything.
    """
    pass
```

- The first line typically contains usage information (starting with an upper-case letter, and terminated with a full stop).
- After that several more paragraphs can be added, explaining details of the usage information.
- This information can be accessed using `.__doc__` or `help` constructs.

```
my_function.__doc__   # return doc string
help(my_function)     # print  doc string
```

Anonymous Functions

- A function can be passed as an expression to another function:

```
>>> lambda x, y: x    # in ML: fn x => fn y => x
<function <lambda> at 0xb77900d4>
```

- This is a factory-pattern for a function incrementing a value:

```
def make_incrementor(n):
    return lambda x: x + n
                       # in ML: fn x => x+n
f = make_incrementor(42)
f(0)
f(1)
```

Anonymous Functions

Functions are compared using the address of their representation in memory:

```
>>> (lambda x:x)==(lambda x:x)  # Make 2 functions, compare addres
False                           # Different!
>>> x = (lambda x:x)            # Make 1 function
>>> x == x                      # compare its address with itself.
True                            # The same!
```

What does this return?

```
>>> x = [lambda x:x]*2
>>> x[0]==x[1]
True
```

This tells us that $*2$ generates a link rather than trigger a copy.

Exercises

- Implement Euclid's greatest common divisor algorithm as a function over 2 int parameters.
- Implement matrix multiplication as a function taking 2 2-dimensional arrays as arguments.

More list operations

- Modifiers:
  - `l.extend(l2)` means `l[len(l):] = l2`, i.e. add `l2` to the end of the list `l`.
  - `l.remove(x)` removes the first instance of `x` in `l`. Error, if `x not in l`.
- Read-only:
  - `l.index(x)` returns the position of `x` in `l`. Error, if `x not in l`.
  - `l.count(x)` returns the number of occurrences of `x` in `l`.
  - `sorted(l)` returns a new list, which is the sorted version of `l`.
  - `reversed(l)` returns an iterator, which lists the elements in `l` in reverse order.

Usage of lists

- Lists can be used to model a *stack*: `append` and `pop()`.
- Lists can be used to model a *queue*: `append` und `pop(0)`.

Higher-order functions on lists

- `filter(test, sequence)` returns a sequence, whose elements are those of `sequence` that fulfill the predicate `test`. E.g.

  ```
  filter(lambda x: x % 2 == 0, range(10))
  ```

- `map(f, sequence)` applies the function `f` to every element of `sequence` and returns it as a new sequence.

  ```
  map(lambda x: x*x*x, range(10))
  map(lambda x,y: x+y, range(1,51), range(100,50,-1))
  ```

- `reduce(f, [a1,a2,a3,...,an])` computes `f(...f(f(a1,a2),a3),...,an)`

  ```
  reduce(lambda x,y:x*y, range(1,11))
  ```

- `reduce(f, [a1,a2,...,an], e)` computes `f(...f(f(e,a1),a2),...,an)`

List comprehensions

- More readable notation for combinations of `map` and `filter`.
- Motivated by *set comprehensions* in mathematical notation.
- `[ e(x,y) for x in seq1 if p(x) for y in seq2 ]`

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
```

Deletion

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]          # Kill first element
>>> a
[1, 66.25, 333, 333, 1234.5] # Gone!
>>> del a[2:4]        # Kill 3rd + 4th elements
>>> a
[1, 66.25, 1234.5]  # Gone!
>>> del a[:]          # Kill them all!!
>>> a                 # (Who says megalomania can't be fun?)
[]                    # Gone!
>>> del a             # Now ... kill the var itself.
>>> a
NameError: name 'a' is not defined # What var?
```

Tuples

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> x, y, z = t
>>> empty = ()
>>> singleton = 'hello',    # trailing comma
```

Sets

- `set(l)` generates a set, formed out of the elements in the list `l`.
- `set(l)` generates a frozenset (an immutable set), formed out of the elements in the list `l`. More on this in Slide 71.
- `list(s)` generates a list, formed out of the elements in the set `s`.
- `tuple(s)` generates a tuple (an immutable list), formed out of ...you know the rest.
- `x in s` tests for set membership
- Operations: `-` (difference), `|` (union), `&` (intersection), `^` (xor).
- `for v in s` iterates over the set (sorted!).

Sets

Warning: `set` and `list` treat string arguments as lists of chars.

```
>>> x = "Supercalifragilisticexpialidocious"
>>> list(x)
['S', 'u', 'p', 'e', 'r', 'c', 'a', 'l', 'i', 'f', 'r', 'a',
 'g', 'i', 'l', 'i', 's', 't', 'i', 'c', 'e', 'x', 'p', 'i',
 'a', 'l', 'i', 'd', 'o', 'c', 'i', 'o', 'u', 's']
>>> tuple(x)
('S', 'u', 'p', 'e', 'r', 'c', 'a', 'l', 'i', 'f', 'r', 'a',  ...
>>> set(x)       # Sets can't have repeated elements
{'S', 'i', 'c', 't', 'u', 'x', 'e', 's', 'a', 'r', 'g', 'd',
 'l', 'f', 'o', 'p'}
>>> list([x])    # We probably meant this
['Supercalifragilisticexpialidocious']
>>> tuple([x])
('Supercalifragilisticexpialidocious',)
>>> set([x])
{'Supercalifragilisticexpialidocious'}
```

Dictionaries

Dictionaries are finite maps, *hash maps*, *associative arrays*. They represent unordered sets of (key, value) pairs.

```
tel = {'jack': 4098, 'guido': 4127}            # These are
tel = dict([('guido', 4127), ('jack', 4098)])  # equivalent
print(tel)
# output: {'jack': 4098, 'guido': 4127}
print(tel['jack'])                    # Access is through the key
# output: 4098
tel['jack'] = 4099
print(tel)
# output: {'jack': 4099, 'guido': 4127}
tel = {'jack': 4098, 'jack': 4099 } # A key may occur at most once
print(tel)
# output: {'jack': 4099}
```

Deletion + more

```
tel = {'jack':4098}
del['jack']          # Delete key once
print(tel)           # output: {}
del['jack']          # Delete now-non-existent key: Error
tel = {'jack': 4098, 'guido': 4127}
```

More fun:

```
>>> tel.keys()
dict_keys(['jack', 'guido'])
>>> tel.values()
dict_values([4098, 4127])
>>> tel.items()
dict_items([('jack', 4098), ('guido', 4127)])
>>> list(tel.items())  # Why the 'list' wrapper here?
[('jack', 4098), ('guido', 4127)]
>>> list(enumerate(tel))
[(0, 'jack'), (1, 'guido')]
```

Mutable vs immutable types

Python distinguishes between *mutable* and *immutable* types. Mutable types have methods to change the value; immutable types don't.
See https://docs.python.org/3.0/library/stdtypes.html#typesseq-mutable onwards.
You are expected to know that dictionary hashes must be immutable, and know which types are mutable and which aren't (and so can't hash in dictionaries).
Lists are mutable, strings and tuples are immutable. Thus:

```
>>> tel = { 'jack': 1234 }       # Fine
>>> tel = { ('jack',): 1234 }    # Fine
>>> tel = { ['jack']: 1234 }     # Not fine
TypeError: unhashable type: 'list'
```

Sets are mutable, frozensets are immutable. Thus:

```
>>> tel = { {'jack'}: 1234 }                  # Not fine
>>> tel = { frozenset(['jack']): 1234 }       # Fine
```

Mutable vs immutable types

`set` and `list` have add/append; `frozenset` and `tuple` don't:

```
>>> x = set([])
>>> x.add("*")
>>> x
{'*'}
>>> x = frozenset([])
>>> x.add("*")
AttributeError: 'frozenset' object has no attribute 'add'
>>> x = list([])     # Or just x=[]
>>> x.append("*")
>>> x
['*']
>>> x = tuple([])    # Or just x=()
>>> x.append("*")
AttributeError: 'tuple' object has no attribute 'append'
```

Warning: `x={}` creates an empty dictionary (not an empty set).

Loop techniques

- Here are some useful patterns involving loops over dictionaries.
- Simultaneous iteration over both keys and elements of a dictionary:

```
l = ['tic', 'tac', 'toe']
for i, v  in enumerate(l):
    print (i, v)
```

- Simultaneous iteration over two or more sequences:

```
for i, v in zip(range(len(l)),l):
    print (i, v)
```

- Iteration in sorted and reversed order:

```
for v in reversed(sorted(l)):
  print (v)
```

Comparison of sequences and other types

- Sequences are compared lexicographically, and in a nested way:

```
() < ('\x00',)
('a', (5, 3), 'c') < ('a', (6,) , 'a')
```

- Warning: The comparison of values of *different* types doesn't produce an error. It returns an arbitrary value!

```
>>> "1" < 2
False
>>> () < ('\x00')
False
>>> [0] < (0,)
True
```

Equality and identity in Python

Equality is `x == y`. Seems simple—but it isn't. Equality is just a method; we can program 'equality' to do anything at all:

```
class Equal(): # class equal to everything (thanks to Jared Grubb)
    def __eq__(self, other):
        print("Equality test alert!!!")
        return True
```

So watch this:

```
>>> Equal() == 5
Equality test alert!!!
True
>>> Equal() == 6
Equality test alert!!!
True
```

Worse, it is possible to silently overwrite the equality methods of familiar datatypes. You cannot be sure what code is actually being called when you call ==.

Equality and identity in Python

Python has a primitive `is`; tests *identity*, cannot be overwritten:

```
>>> Equal() is 7
False
```

This seems simple, right? No, it isn't. Functions are compared by address not mathematical equality (cf Slide 59):

```
>>> (lambda x:x) == (lambda x:x)
False
>>> (lambda x:x) is (lambda x:x)
False
```

Equality and identity in Python

Also, Python may link substructures rather than copy them:

```
>>> x = [[]]
>>> y = x*2
>>> z = [[],[]]
>>> y[0]==y[1]
True
>>> y[0] is y[1]
True
>>> z[0]==z[1]
True
>>> z[0] is z[1]  # What's happened here?
False
```

`*2` creates a link; it doesn't copy. So `y[0]` and `y[1]` are indeed *identical*.

You are expected to understand this. So for instance . . .

Equality and identity in Python

```
>>> x = [[]]*3
>>> x
[[],[],[]]
>>> x[0].append("Ha")
>>> x
[['Ha'],['Ha'],['Ha']]
>>> x = [[],[],[]]
>>> x[0].append("Ha")
>>> x
[['Ha'],[],[]]
```

I understand the logic: Python is just being efficient. But personally, this gives me the heebie-jeebies.

Equality and identity in Python

You are expected to understand what's happening here:

```
>>> x = []
>>> y = []
>>> x == y
True
>>> x is y
False
>>> x.append("Gotcha")
>>> y
[]
```

```
>>> x = []
>>> y = x
>>> x == y
True
>>> x is y
True
>>> x.append("Gotcha")
>>> y
['Gotcha']
```

Be careful!

Copy and deepcopy

Python, of course, has a library for this.

```
>>> import copy
>>> l = [[]]
>>> m = copy.copy(l)
>>> n = copy.deepcopy(l)
>>> l[0].append("Hi")
>>> l
[['Hi']]
>>> m
[['Hi']]
>>> n
[[]]
```

So `copy.copy` makes a copy with links to original structure; `copy.deepcopy` makes a 'true' copy ...

Copy and deepcopy

. . . so let's have some innocent fun with this.

```
def biglist(n): # create *a lot* of copies of [].
    if n==0:
        return [[]];
    else:
        return biglist(n-1)*(n**n)
```

`biglist(n)` increases as $O(n!^2)$.

```
l = biglist(5)
m1 = l                    # very fast
m2 = copy.copy(l)         # quite fast
m3 = copy.deepcopy(l) # rather slow
```

You are expected to anticipate what happens to `m1`, `m2`, and `m3` if we execute `l[0].append('Hi')`.

More on recursion

You are expected to know:

- *Recursion* is when a program calls itself. See the notes on ML.
- *Tail-recursion* is when a program calls itself and this is the end of that program's computation. Therefore, the local state of the program does *not* need to be stored, and indeed, it can be re-used.
  Tail-recursion is, in effect, equal to iteration.

For more reading I recommend this webpage on recursion:

`python-course.eu/python3_recursive_functions.php`

More on recursion

Let's look at four implementations of Fibonacci:

```
def fibi(n):    # iterative
 a, b = 1, 1
 for i in range(n):
  a, b = b, a+b
 return(a)
```

```
def fibr(n):    # recursive
# print(n,end=' ')
 if n==0 or n==1:
  return(1)
 else:
  return fibr(n-1)+fibr(n-2)
```

```
memo = { 0:1, 1:1 } # memoised
def fibm(n):
 if not n in memo:
  memo[n] = fibm(n-1)+fibm(n-2)
 return memo[n]
```

```
def fibg(n): # with static local
 if not hasattr(fibg, "memo"):
  fibg.memo = { 0:1, 1:1 }
 if not n in fibg.memo:
  fibg.memo[n]=fibg(n-1)+fibg(n-2)
 return fibg.memo[n]
```

More on recursion

The iterative `fibi` works fine.
The recursive `fibr` works fine but *slowwww* because: it calls itself recursively twice and recalculates values many times (the call to `n-1` will recursively call `n-2`, again). You are expected to understand this.
Uncomment `print(n)` in the `fibr` code and try `fibr(20)`.
The memoised `fibm` is much faster. You are expected to be able to explain why.
The memoised `fibg` is faster and uses some (non-examinable) fancy programming to keep `memo` local.

More on recursion

The catch: Python has a global limit of 999 on recursive calls.

```
>>> fibm(10)   # works
89
>>> fibm(1010) # doesn't work
RuntimeError: maximum recursion depth exceeded
>>> fibm(50)   # works
20365011074
>>> fibm(1010) # why does it work the second time?
8650063399098190712106206706196570348687189343891375136228331652681264351506724942569273596220431478591685532601934918119485119259903262937328348963115101043233518517503077734954246670528164366878561010490499714
```

This is pants. Python doesn't implement recursion; it *pretends* to. This makes me angry.

More on recursion

A justification is that CPython doesn't optimise tail recursion to re-use the existing stack, so recursive calls in Python are quite memory-hungry.
If we're determined we can work around this, e.g.
http://code.activestate.com/recipes/474088/
But (IMO) Python should do this automatically.

Modules

- Every Python file is a module.
- `import myMod` imports module `myMod`.
- The system searches in the current directory and in the `PYTHONPATH` environment variable.
- Access to the module-identifier `x` is done with `myMod.x` (both read and write access!).
- The code in the module is evaluated, when the module is imported the first time.
- Import into the main name-space can be done by

Example

```
from myMod import myFun
from yourMod import yourValue as myValue

myFun(myValue) # qualification not necessary
```

- **NB:** In general it is not advisable to do `from myMod import *`.

Output formatting

- `str(v)` generates a “machine-readable” string representation of `v`
- `repr(v)` generates a representation that is readable to the interpreter. Strings are escaped where necessary.
- `s.rjust(n)` fills the string, from the left hand side, with space characters to the total size of `n`.
- `s.ljust(n)` and `s.center(n)`, analogously.
- `s.zfill(n)` inserts zeros to the number `s` in its string representation.
- `'-3.14'.zfill(8)` yields `'%08.2f' % -3.14`.
- Dictionary-Formating:

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098 }
>>> print ('Jack: %(Jack)d; Sjoerd: %(Sjoerd)d' % table)
Jack: 4098; Sjoerd: 4127
```

Exceptions

- Exceptions can be caught using a `try...except...` expression.

Example

```
while True:
  try:
    x = int(raw_input("Please enter a number: "))
    break
  except ValueError:
    print ("Not a valid number.  Try again...")
```

- It is possible to catch several exceptions in one `except` block:

```
except (RuntimeError, TypeError, NameError):
  pass
```

Exceptions

- Several exception handling routines

Example

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print ("I/O error(%s): %s" % (errno, strerror))
except ValueError:
    print ("Could not convert data to an integer.")
except:
    print ("Unexpected error:", sys.exc_info()[0])
    raise
```

Exceptions: else

- If no exception was raised, the optional `else` block will be executed.

Example

```
for arg in sys.argv[1:]:
  try:
    f = open(arg, 'r')
  except IOError:
    print ('cannot open', arg)
  else:
    print (arg, 'has', len(f.readlines()), 'lines')
    f.close()
```

Raising Exceptions

- `raise Ex[, info]` triggers an exception.
- `raise` triggers the most recently caught exception again and passes it up the dynamic call hierarchy.

```
>>> try:
...     raise NameError, 'HiThere'
... except NameError:
...     print ('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

Clean-up

- The code in the `finally` block will be executed at the end of the current `try` block, no matter whether execution has finished successfully or raised an exception.

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print ('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

Exceptions: All Elements

- Here is an example of an `try` constructs with all features:

Example

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print ("division by zero!")
    else:
        print ("result is", result)
    finally:
        print ("executing finally clause")
```

Pre-defined clean-up

- `with` triggers automatic clean-up if an exception is raised
- In the example below, the file is automatically closed.

Example

```
with open("myfile.txt") as f:
    for line in f:
        print (line)
```

- Using `with` is good style, because it guarantees that there are no unnecessary, open file handles around.

Iterators in detail

- `it = iter(obj)` returns an iterator for the object `obj`.
- `it.next()` returns the next element
- or raises a `StopIteration` exception.

Generators

- A method, containing a `yield` expression, is a **generator**.

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

- Generators can be iterated like this.

```
>>> for char in reverse('golf'): print (char)
...
f l o g
```

Generator Expressions

- Similar to list-comprehensions:

```
>>> sum(i*i for i in range(10))
285
>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))
260
>>> unique_words = set(word
                       for line in page
                       for word in line.split())
```

Exercises

- Go to the Python Online Tutor web page, www.pythontutor.com, and do the object-oriented programming exercises (OOP1, OOP2, OOP3).
- Implement the data structure of binary search trees, using classes, with operations for inserting and finding an element.
- Python practice book.
- The python course.

Church-Rosser

Python is not Church-Rosser. Very much not so. Order of evaluation matters.
Python seems to have poor support for parallel execution. There are libraries, but the mathematician in me says they're patching over something that's fundamentally broken.
The whole design philosophy of Python is optimised (it seems to me) for the *single threaded execution model*. Simple, but (obviously) this does not scale to multiple processors or cores.
Python doesn't really manage threading at all; it delegates this to the OS.
This is partly why it is so amusingly easy for Python to crash my machine: if a Python program hogs processor or memory resources, I have to send it an OS interrupt.
Python itself won't step in to throttle or manage its own processes.

The vision

Please, don't just learn Python in this course.
My goal with these notes is twofold:

- To put Python in some kind of perspective with respect to other languages.
- To explore the edge cases of Python's design, such as:
  - Limits on recursion.
  - Implications of shallow copying, etc.
  - (Im)precision of `int` and `float`.
  - Equality and identity and why you can't trust them!
  - Mutable vs immutable types.
  - Anonymous functions, memoisation, (poor) parallelisation, etc.

There's tons more cool stuff out there, but . . .

Everything is non-examinable beyond this point

.

Non-useful links:

- http://uselessfacts.net/
- http://www.freemaninstitute.com/uselessFacts.htm
- http://www.straightdope.com/ (recommended)
- http://www.howstuffworks.com/ (founded by a person called 'Marshall Brain'; couldn't make it up).

Overloading

- Operators such as +, <= and functions such as `abs`, `str` and `repr` can be defined for your own types and classes.

Example

```
class Vector(object):
  # constructor
  def __init__(self, coord):
    self.coord = coord
  # turns the object into string
  def __str__(self):
    return str(self.coord)

v1 = Vector([1,2,3])
# performs conversion to string as above
print (v1)
```

Overloading

Example

```
class Vector(object):
  # constructor
  def __init__(self, coord):
    self.coord = coord
  # turns the object into string: use <> as brackets, and ; as se
  def __str__(self):
    s = "<"
    if len(self.coord)==0:
        return s+">"
    else:
        s = s+str(self.coord[0])
    for x in self.coord[1:]:
        s = s+";"+str(x);
    return s+">"

v1 = Vector([1,2,3]); print (v1)
```

Overloading arithmetic operations

Example

```
import math     # sqrt
import operator # operators as functions

class Vector(object):
  ...
  def __abs__(self):
    '''Vector length (Euclidean norm).'''
    return math.sqrt(sum(x*x for x in self.coord))
  def __add__(self, other):
    '''Vector addition.'''
    return map(operator.add, self.coord, other.coord)

print(abs(v1))
print(v1 + v1)
```

Overloading of non-symmetric operations

- Scalar multiplication for vectors can be written either `v1 * 5` or `5 * v1`.

Example

```
class Vector(object):
  ...
  def __mul__(self, scalar):
    'Multiplication with a scalar from the right.'
    return map(lambda x: x*scalar, self.coord)

  def __rmul__(self, scalar):
    'Multiplication with a scalar from the left.'
    return map(lambda x: scalar*x, self.coord)
```

- `v1 * 5` calls `v1.__mul(5)`.
- `5 * v1` calls `v1.__rmul(5)`.

Overloading of indexing

- Indexing and segment-notation can be overloaded as well:

Example

```
class Vector(object):

  def __getitem__(self, index):
    '''Return the coordinate with number index.'''
    return self.coord[index]

  def __getslice__(self, left, right):
    '''Return a subvector.'''
    return Vector(self.coord[left:right])

print v1[2]
print v1[0:2]
```

Exercise (optional)

- Define a class `Matrix` and overload the operations `+` and `*` to perform addition and multiplication on matrices.
- Define further operations on matrices, such as `m.transpose()`, `str(m)`, `repr(m)`.

Interpretation

- Strings can be evaluated using the function `eval`, which evaluates string arguments as Python expressions.

```
>>> x = 5
>>> eval ('x')
5
>>> f = lambda x: eval('x * x')
>>> f(4)
16
```

- The command `exec` executes its string argument:

```
>>> exec 'print x'
5
```

Compilation

- The following command performs compilation of strings to byte-code:

```
c = compile('map(lambda x:x*2,range(10))', # code
  'pseudo-file.py',      # filename for error msg
  'eval')  # or 'exec' (module) or 'single' (stm)
eval(c)
```

- Beware of indentation in the string that you are composing!

```
>>> c2 = compile('''
... def bla(x):
...   print x*x
...   return x
... bla(5)
... ''', 'pseudo', 'exec')
>>> exec c2
25
```

Selected library functions

- One of the main reasons why Python is successful is the rich set of libraries
- This includes standard libraries, that come with a Python distribution, but also third-party libraries
- Prominent third-party libraries are:
  - JSON
  - matplotlib
  - tkinter
  - numpy
  - scipy
  - sympy
  - orange
  - pandas

String libraries and regular expressions

- Python, as many scripting languages, has powerful support for **regular expressions**
- Regular expression can be used to search for strings, replace text etc
- The syntax for regular expression is similar across languages
- For working experience with regular expressions, see this section of the Linux Introduction or these slides on regular expressions.
- There are many good textbooks on regular expressions around.

Basic usage of string libraries and regular expressions

- To **access** the regular expression library use: `import re`
- To **search** for a `substr` in `str` use: `re.search(substr,str)`
- To **replace** a `pattern` by a `repstr` in `string` use: `re.sub(pattern, repstr, string)`
- To **split** a `string`string into `sep`-separated components use: `re.split(pattern,string)`
- Check the Python library documentation for details and more functions.

Examples of regular expressions in Python

Read from a file, print all lines with 'read' event types:

Example

```
file='/home/hwloidl/tmp/sample_10k_lines.json'
print ("Reading from ", file)
with open(file,"r") as f:
    for line in f:
        if (re.search('"event_type":"read"', line)):
            print (line)
```

Pick-up the code from the sample sources section

Examples of regular expressions in Python

Read from a file, split the line, and print one element per line

Example

```
file='/home/hwloidl/tmp/sample_10k_lines.json'
print ("Reading from ", file)
with open(file,"r") as f:
    for line in f:
        if (re.search('"event_type":"read"', line)):
            line0 = re.sub("[{}]", "", line)     # remove {}
            for x in re.split("[ ]*,[ ]*",line0):# split by ','
                print (re.sub(':','->', x))      # replace ':' by
```

Saving structured data with JSON

- JSON (JavaScript Object Notation) is a popular, light-weight data exchange format.
- Many languages support this format, thus it's useful for data exchange across systems.
- It is much ligher weight than XML, and thus easier to use.
- `json.dump(x, f)` turns `x` into a string in JSON format and writes it to file `f`.
- `x = json.load(f)` reads `x` from the file `f`, assuming JSON format.
- For detail on the JSON format see: `http://json.org/`

JSON Example

Example

```
tel = dict([('guido', 4127), ('jack', 4098)])
ppTelDict(tel)

# write dictionary to a file in JSON format
json.dump(tel, fp=open(jfile,'w'), indent=2)
print("Data has been written to file ", jfile);

# read file in JSON format and turn it into a dictionary
tel_new = json.loads(open(jfile,'r').read())
ppTelDict(tel_new)

# test a lookup
the_name = "Billy"
printNoOf(the_name,tel_new);
```

Visualisation using `matplotlib`

`matplotlib` is a widely used library for plotting data in various kinds of formats. Advantages of the library are

- It supports a huge range of graphs, such as plots, histograms, power spectra, bar charts, errorcharts, scatterplots etc
- It provides interfaces to external tools such as MATLAB
- It is widely used and well-documented
- For detailed documentation see: Matplotlib documentation

Examples of using `matplotlib`

The following code displays a histogram in horizontal format, with hard-wired data:

Example

```
import matplotlib.pyplot as plt
...
# # horizontal bars: very simple, fixed input
plt.barh([1,2,3], [22,33,77], align='center', alpha=0.4)
#         indices  values
plt.show()
```

Pick-up the code from Sample sources (simple_histo.py)

A basic GUI library for Python: `tkinter`

- `tkinter` is a basic library for graphical input/output
- It has been around for a long time, and is well supported
- It uses the Tcl/TK library as backend
- It features prominently in textbooks such as:

  Mark Lutz, *"Programming Python."* O'Reilly Media; 4 edition (10 Jan 2011). ISBN-10: 0596158106.
- For details and more examples see: tkinter documentation

For examples see Sample Sources (feet2meter.py)

Some more libraries

- Sage. Mathematics software system licensed under the GPL. Supports GAP, Maxima, FLINT, R, MATLAB, NumPy, SciPy, matplotlib, etc. Python is **glueware**; the (heavy) computation is done in the external libraries.
- Numpy. Library of mathematical/scientific operations with arrays, linear algebra, Fourier transform, random numbers, and integration with C(++) & Fortran.
- Orange. Python library for **data analytics, data visualisation and data mining.**
- pandas. Python data analysis toolkit. Provides functions for constructing frames that can be accessed and manipulated like database tables. Similar in spirit to C#'s LINQ sub-language. Focus is on *data manipulation*, not on statistics or scientific computing.
- SymPy. Python library for **symbolic mathematics**.

Further reading

- Mark Lutz, *"Programming Python."*
  O'Reilly Media; 4 edition (10 Jan 2011). ISBN-10: 0596158106.
- Wes McKinney, *"Python for data analysis"[eBook]*
  O'Reilly, 2013. ISBN: 1449323626
  Focus on libraries for data-analytics.
- Hans Petter Langtangen, *"A Primer on Scientific Programming with Python"* 4th edition, 2014. ISBN-10: 3642549586
  Focussed introduction for scientific programming and engineering disciplines.
- Drew A. McCormack *"Scientific scripting with Python."*
  ISBN: 9780557187225
  Focussed introduction for scientific programming and engineering disciplines.